ECOLE
POLYTECHNIQUE
DE BRUXELLES

COMMUNICATION NETWORKS :
PROTOCOLS AND ARCHITECTURES

ELEC-H417

PROJECT

Authors
Haythem BOUGHARDAIN
Yasser FRAJI
Reda MEFTAH

Professor
Jean-Michel DRICOT

Assistants
Wilson DAUBRY
Denis VERSTRAETEN

ACADEMIC YEAR 2021–2022

CONTENTS

1 INTRODUCTION

As part of the project in the COMMUNICATION NETWORKS: PROTOCOLS AND ARCHITECTURES course, students are asked to design and implement a basic chat app enabling private communication, based on a central server allowing a number of clients to create an account and engage in conversations with each other.

This report will go over the architecture choices for the coding the project, the features added on top of the required points, as well as the difficulties faced when implementing the application.

2 ARCHITECTURE

First of all, the programming language we decided to use for this project was Java, for several reasons: it is a language we have already used to create applications with a graphical user interface, and it allows easy object-oriented programming, meaning we can easily implement a Model-View-Controller design pattern to structure our code as efficiently as possible.

Thus, the main architecture of our project consists of separating classes in three groups : the model classes, which define the business logic of the whole application; the view classes, which are linked to the graphical aspect of the program, and which call the relevant methods when an action is carried out in the GUI; and the controller classes, which link the models and the views, making sure that the action events in the view produce the correct sequence of responses from the model.

As is asked in the project's statement, a client and a server have been implemented: the client will benefit of a GUI so that a user can use it as easily as possible, while the server simply consists of a program which will run in the background, and can be executed by the user himself if he wants to host his own server.

In the rest of this section, we will go through the different groups of the MVC design pattern for the client application and explain what each of the classes does. We will delve into the network aspects of the client- and server-related classes which are defined in the model.

2.1 Views

To implement a GUI in Java, the JavaFX library is used. Since it is packaged by default with Java 1.8, using this version allows us to directly be able to define our GUI without importing foreign libraries.

JavaFX comes with its FXML interpreter, which can read and interpret `.fxml` files that define the different scenes of our application. In our case, three different scenes are defined:

- `LogIn`: This scene serves as the home page of the program by default, and lets the user type their username and password in dedicated textfields. If the user does not have an account yet, they can create one by following the hyperlink to reach the `Register` scene.

- `Register`: Built in the same manner as the `LogIn` scene, it allows a user to create their own account using their first and last names, their username, their email address, and a password. When succesfully creating an account, they are sent back to the `LogIn` scene.

- `Messenger`: This scene is the main page users will want to use when launching the program. It allows to see all other connected users in the pane on the left, and to type in or read messages thanks to the large text box. In all cases, the sender of each message is specified so that the user knows who they are talking to. The user can also disconnect, logging them out of the program and sending them back to the `LogIn` scene.

2.2 Controllers

For each view mentioned above, there is a controller corresponding to the concerned scene in the program. As is the basis of the MVC design pattern, each controller has its corresponding view as an attribute to be able to call its methods easily, while a view defines an interface implemented by the controller to declare the latter's methods. It is unnecessary to go into too much detail concerning each controller, as they all simply use methods defined in the model or view classes.

Before going into the models however, it is important to remember the `ClientMain` class manages which controller to call and at what time, and can technically be considered the controllers' controller. This is done using each of the controllers' `show` method, which allows to easily switch scenes inside the GUI's window.

2.3 Models

We will now go over each of the classes that help define how the network is managed, and the classes which define user management, all in successive order as has been done in Subsection 2.1:

2.3.1 Network management

- `Client`: This class is defined as a singleton to be easily accessible by all the other classes in the program, and to be sure that a single client application will generate a single client connecting to a server. The connection to the server is done using a socket defined by the server's name (it is simply the IP address of the server in this case) and the adequate server port (defined arbitrarily as `2023` here). To read incoming requests sent by the server, a parallel thread is created, which loops infinitely while checking the client's input stream: when a request arrives, the client takes the appropriate action. Such requests are defined by two listener interfaces, `UserStatusListener` and `MessageListener`, which are then implemented by `MessengerController`: these allow the client to tell the controller when another user goes online or offline, and when a message is received, respectively. Thanks to the manually imported `StringUtils` class from Apache, it is possible to easily parse incoming requests. Using the client's output stream, it is also possible to send several requests: sending a message to a specific user, logging in, creating an account, and disconnecting. Finally, a hash function is defined in this class to notably hash passwords, to ensure that the password a user uses to log in or register are not plainly sent to the server.

- `Server`: This class defines the server's simple behaviour, which is simply to initialise the database which will contain all the users, as well as executing an infinite loop to accept all incoming client connections. In the case a client connects, this class creates a new `ServerThread`, which will be explained in more detail below.

- `ServerThread`: This class extends the Java `Thread` class, which means that several instances of this class can be run in parallel: this is done to ensure that the central server can manage and listen to all incoming connections from clients. The main loop for this class is defined in the `handleSocket` method, which, as seen in the `Client` class, will indefinitely read all requests arriving from a client: these include the same as those mentioned when talking about the requests a client can send. Naturally, all of these requests have their own method to handle each of these cases, at the end of which a message is sent to the client to let them know the outcome of the request (e.g. sending `ok register` if the creation of an account was successful, or `error register` if it was not).

2.3.2 User management

- `User`: This class is simple in its conception and allows the creation of users according to their first name, last name, username, email address and password. Most of the methods defined are getters and setters, apart from `checkSyntax` which, as its name suggests, checks that all relevant fields are properly defined. This allows to easily check if a user can be created with these parameters.

- `Database`/`UserDatabase`: `UserDatabase` extends `Database`, the latter being an abstract class allowing to serialise any type of data, while the former overrides some of the methods defined in its super-class to specialise in the serialisation of `User` objects. It also comes with methods which can find specific users in the database thanks to their username or email address, for example.

All in all, by looking at all the classes defined for the project, it is possible to understand the global functioning of the developed application:

1. The server is launched thanks to the `ServerMain` class: its IP address will be the IP address of whatever computer or terminal it is launched from, and the port number is arbitrarily defined as `2023`.

2. The client is launched by using the `ClientMain` class: a socket is created to link it to the server, which will generate a `ServerThread` to manage the client concerned.

3. Any requests from the user through the client (such as logging in or sending a message) are relayed to the server, which takes the appropriate action, and can send a confirmation message back to the client in certain cases.

3 INNOVATION & CREATIVITY

First of all, we decided to give a name to our application, Mingle, as it allows user to do exactly that: mingle. Users can simply launch the application and see all other connected users, and decide to chat with any one of these. By definition, our implementation ensures that only users that are simultaneously online can chat with each other. This allows Mingle to act as a friend-based social network, since someone can still choose to send messages to their friends, but Mingle can also as a stranger-meeting service, since anyone can strike a conversation with any other user.

Furthermore, the creation of a graphical user interface allows the application to be much more accessible to everyday users: all they need to do is launch the client program, create an account, log in, and they can chat with any other user they wish. This is translated well in the GUI for the messenger scene, as shown in Figure 1 below. We can see two main parts in this scene: the pane on the left shows the usernames of all other connected users, and allows the user to choose to which person they would like to send a message (selecting a user in the list defines that the following messages will be sent to them, while not selecting anyone logically stops the user from sending any message). The large text box contains the history of all the conversations which include the user concerned, this means all sent and received messages.

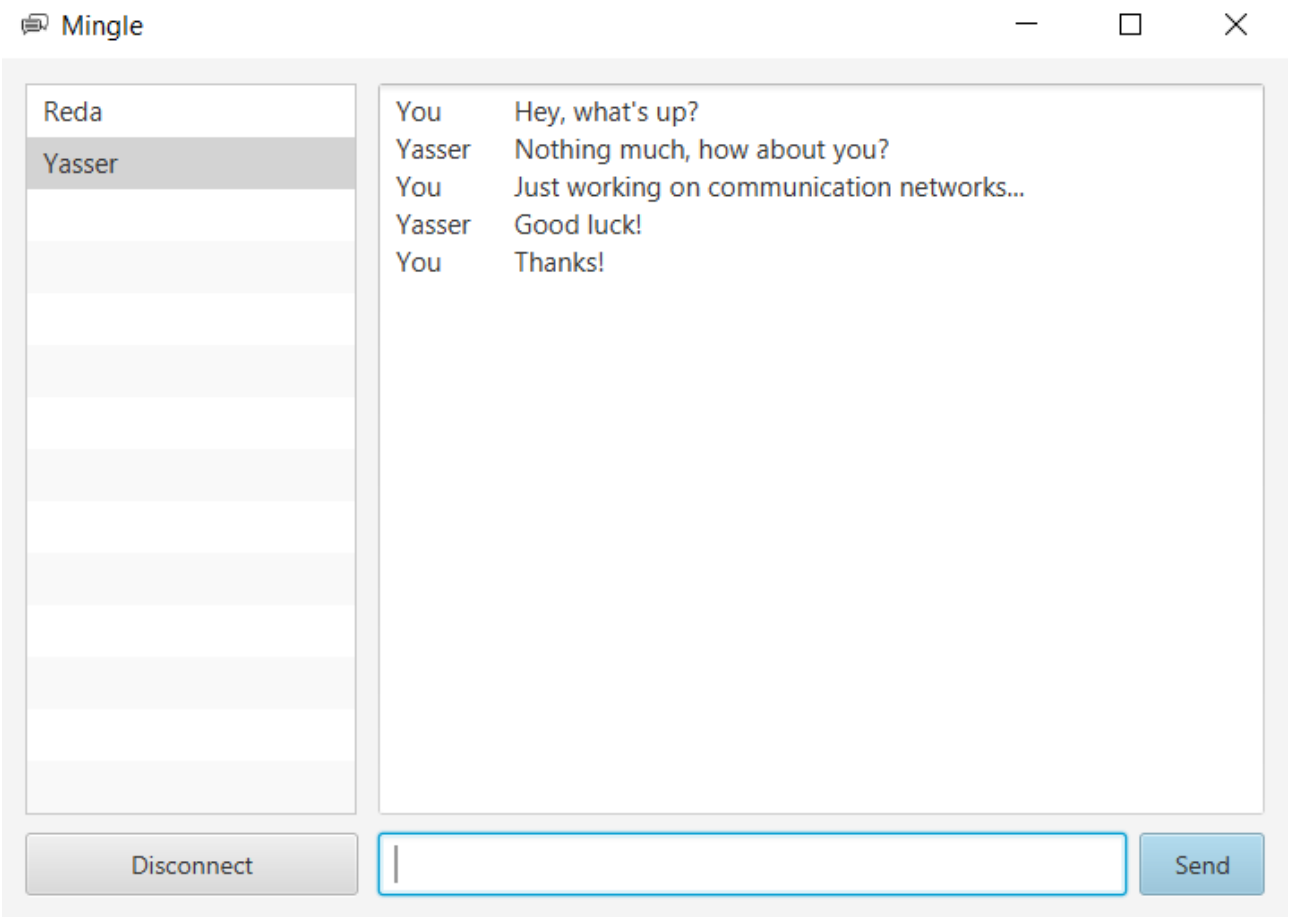

Figure 1: Screenshot of the messenger scene

4 CHALLENGES

Many challenges were encountered when confronting this project. The most prominent was translating what was taught in the theoretical courses about networking and server-client interactions into a practical application. Indeed, understanding these concepts does not necessarily mean that applying these in a specific programming language is easy.

To first understand a good architecture of how a server and a client work, it was important to take a look online to understand how these interactions were coded. One particular online guide from *FullStackMastery* allowed us to understand how sockets were used in Java to

pass messages from a client to a server, and parts of the proposed architecture remain implemented in the same manner in our application.

Implementing a graphical user interface in Java can also pose a challenge in this sort of projects, but thanks to the previously followed INFO-F307 course where our main project was doing exactly that, we already had salvageable code for the whole MVC pattern design, as well as the user management classes such as `User` and `Database`.

Another important difficulty we faced was concerning the encryption of messages sent between client and server. Indeed, it was quite difficult to find first-timer-friendly documentation online about the implementation of private communication, and most of the code snippets proposed online would not work in our case. Some of these include attempting to use the `SSLSocket` class proposed directly by Java instead of the plain `Socket`, and resulted in stopping all connections with the server. Normally, this should allow the client and server to connect according to the TLS protocol, encrypting messages sent between each other. However, this requires the program creator to generate certificates for the server and the client, and setting a keystore and/or a truststore for each of them. This process is also difficult to grasp for students who had never coded a program related to cryptography before. At the end of the day, we unfortunately did not manage to implement a functioning message encryption system for our application, rendering any communications intercepted insecure and easily readable by a malicious third-party.

5 CONCLUSION

In conclusion, we believe this project was very interesting as it allowed us to create our own application and delve into the more practical aspects of networking, as opposed to simply learning theoretical concepts from courses. Our project resulted in a functioning application allowing clients to connect to a server, in turn allowing users to create an account, log in and chat with other users. However, as mentioned above, encrypting messages sent from one another was unfortunately not carried out in this project. Ultimately, we feel that, for the upcoming years, the project could be more relevant if basics of network and cryptography management in a specific programming language could be taught in order for students not to start their project from scratch.